UNIVERSIDADE DO MINHO

Exercício 2

Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio
(2º Semestre / 2019-2020)

A84003 Beatriz Rocha

Braga,
Junho 2020

Capítulo 1

Resumo

O presente relatório consiste numa explicação e demonstração de uma proposta de resolução do Exercício 2 da unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio.

O objetivo deste exercício é a motivação para a utilização da Programação em Lógica, usando a linguagem de programação PROLOG, no âmbito de métodos de Resolução de Problemas e no desenvolvimento de algoritmos de pesquisa. Neste caso em particular, pretende-se que se desenvolva um sistema, que permita importar os dados relativos às paragens de autocarro, e representá-los numa base de conhecimento e, posteriormente, desenvolver um sistema de recomendação de transporte público para o sistema de transportes do concelho de Oeiras.

Assim sendo, ao longo deste relatório será apresentada a base de conhecimento criada não só com as paragens de autocarros, mas também com as carreiras e ainda o respetivo método de importação dos dados, juntamente com a explicação das alterações que lhes foram feitas. Serão ainda apresentados os mecanismos criados para gerar os grafos necessários à realização das tarefas propostas.

Nas últimas secções serão expostos os predicados criados para responder aos requisitos do sistema. Por fim, serão apresentados os testes realizados por forma a comprovar que o sistema responde corretamente às tarefas pretendidas.

Índice

Lista de Figuras

Capítulo 2

Introdução

A resolução deste exercício teve o intuito de incentivar a utilização da programação em lógica no âmbito da construção de métodos de Resolução de Problemas e no desenvolvimento de algoritmos de pesquisa, recorrendo à linguagem de programação Prolog.

Durante a realização deste exercício foi implementado um sistema de recomendação de transporte público para o sistema de transportes do concelho de Oeiras, que inclui conhecimento de paragens de autocarros, carreiras, arestas e pontos.

Os requisitos deste sistema passam pela implementação de várias funcionalidades que permitam:

- Calcular um trajeto entre dois pontos;
- Selecionar apenas algumas das operadoras de transporte para um determinado percurso;
- Excluir um ou mais operadores de transporte para o percurso;
- Identificar quais as paragens com o maior número de carreiras num determinado percurso;
- Escolher o menor percurso (usando critério menor número de paragens);
- Escolher o percurso mais rápido (usando critério da distância);
- Escolher o percurso que passe apenas por abrigos com publicidade;
- Escolher o percurso que passe apenas por paragens abrigadas;
- Escolher um ou mais pontos intermédios por onde o percurso deverá passar.

Capítulo 3

Preliminares

Com a realização deste exercício pretende-se desenvolver um sistema de recomendação de transporte público para o sistema de transportes do concelho de Oeiras, recorrendo a estratégias de pesquisa informada e não-informada. Assim sendo, torna-se importante distinguir as diferentes características entre ambas as estratégias.

Podemos avaliar o desempenho dos vários algoritmos de pesquisa através dos seguintes critérios:

- Completude, ou seja, se está garantido que encontra a solução;
- Complexidade no tempo, ou seja, quanto tempo demora a encontrar a solução;
- Complexidade no espaço, ou seja, quanta memória necessita para fazer a pesquisa;
- Otimalidade, ou seja, se encontra a melhor solução.

A estratégia de pesquisa informada utiliza o conhecimento específico do problema para dar uma pista para a solução do mesmo. Este tipo de estratégia impede que os algoritmos tropecem no objetivo e na direção da solução. A pesquisa informada pode ser vantajosa em termos de custo, na medida em que a solução ótima é alcançada com custos de pesquisa mais baixos.

Para pesquisar o custo da solução ótima num grafo implementando uma estratégia de pesquisa informada, os nós mais promissores são inseridos numa função heurística. De seguida, a função devolve um número real não negativo, que se trata do custo aproximado do caminho entre esse nó e o nó destino.

O elemento fundamental desta estratégia é a tal função heurística que facilita a transmissão do conhecimento adicional do problema ao algoritmo e, assim, ajuda a encontrar o caminho para o destino através dos nós adjacentes. Entre os vários algoritmos que implementam uma estratégia de pesquisa informada estão a pesquisa gulosa e o algoritmo A*.

A estratégia de pesquisa não-informada (ou cega) difere da pesquisa informada, na medida em que apenas fornece a definição do problmea, mas não há mais nenhum passo para encontrar a solução do mesmo. O principal objetivo da pesquisa não-informada é diferenciar o estado alvo do não-alvo, ignorando o destino para o qual se está a dirigir até descobrir a meta e relatar o sucessor. Entre os vários algoritmos que implementam esta estratégia estão a pesquisa em largura (breadth-first), a pesquisa em profundidade (depth-first) e a pesquisa bidirecional.

	Usa o conhecimento para encontrar a solução	**É mais eficiente**	**Consome mais tempo e custo**
Pesquisa informada	Sim	Sim	Não
Pesquisa não-informada	Não	Não	Sim

Tabela 3.1: Pesquisa informada vs Pesquisa não-informada

Podemos ver as vantagens e desvantagens entre as duas estratégias de pesquisa no quadro acima.

Capítulo 4

Descrição do Trabalho e Análise de Resultados

4.1 Importação dos dados para a base de conhecimento

4.1.1 Paragens de autocarros

No âmbito da importação dos dados relativos às paragens de autocarros do sistema de transportes do concelho de Oeiras que foram fornecidos para a base de conhecimento, comecei por exportar o respetivo ficheiro Excel para um ficheiro CSV.

De seguida, procedi à aplicação de uma série de comandos que corrigiram o CSV para UTF-8, seguidos da aplicação de uma série de comandos que retiraram acentos e cedilhas. Posto isto, executei um *script* que não só eliminou as linhas com campos a nulo, mas também criou a base de conhecimento com as paragens de autocarros que se encontram definidas abaixo:

paragem: Gid, Latitude, Longitude, Estado de Conservação, Tipo de Abrigo, Abrigo com Publicidade, Operadora, Carreira, Código de Rua, Nome da Rua, Freguesia

```
paragem('499',-103758.44,-94393.36,'Bom','Fechado dos Lados','Yes','Vimeca',[01],'300.0','Avenida dos Cavaleiros','Carnaxide e Queijas').
paragem('494',-106803.2,-96265.84,'Bom','Sem Abrigo','No','Vimeca',[01],'389.0','Rua Sao Joao de Deus','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
paragem('480',-106757.3,-96240.22,'Bom','Sem Abrigo','No','Vimeca',[01],'389.0','Rua Sao Joao de Deus','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
paragem('957',-106911.18264993648,-96261.15727273724,'Bom','Sem Abrigo','No','Vimeca',[01],'399.0','Escadinhas da Fonte da Maruja','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
paragem('366',-106021.37,-96684.5,'Bom','Fechado dos Lados','Yes','Vimeca',[01],'411.0','Avenida Dom Pedro V','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
paragem('365',-106016.12,-96673.87,'Bom','Fechado dos Lados','Yes','Vimeca',[01],'411.0','Avenida Dom Pedro V','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
paragem('357',-105236.99,-96664.4,'Bom','Fechado dos Lados','Yes','Vimeca',[01],'1279.0','Avenida Tomas Ribeiro','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
paragem('336',-105143.57,-96690.32,'Bom','Fechado dos Lados','Yes','Vimeca',[01],'1279.0','Avenida Tomas Ribeiro','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
paragem('334',-105336.07,-96668.68,'Bom','Fechado dos Lados','Yes','Vimeca',[01],'1279.0','Avenida Tomas Ribeiro','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
```

Figura 4.1: Excerto da base de conhecimento com as paragens de autocarros

4.1.2 Carreiras

Já no âmbito da importação dos dados relativos às carreiras do sistema de transportes do concelho de Oeiras que foram fornecidos para a base de conhecimento, comecei por comprimir as várias folhas de cálculo do ficheiro Excel numa só e só depois é que exportei o mesmo para um ficheiro CSV.

De seguida, procedi aos mesmos passos referidos anteriormente, ou seja, apliquei uma série de comandos que corrigiram o CSV para UTF-8 e, mais tarde, apliquei uma série de comandos que retiraram acentos e cedilhas. Posto isto, executei um *script* que não só eliminou as linhas com campos a nulo, mas também criou a base de conhecimento com as carreiras que se encontram definidas abaixo:

carreira: Gid, Latitude, Longitude, Estado de Conservação, Tipo de Abrigo, Abrigo com Publicidade, Operadora, Carreira, Código de Rua, Nome da Rua, Freguesia

```
carreira('906',-106791.2,-97137.51,'Bom','Fechado dos Lados','Yes','Carris','776','386','Rua Sacadura Cabral','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
carreira('475',-106826.16,-96699.81,'Bom','Sem Abrigo','No','Carris','776','386','Rua Sacadura Cabral','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
carreira('477',-106835.46,-96672.9,'Bom','Fechado dos Lados','Yes','Carris','776','386','Rua Sacadura Cabral','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
carreira('471',-106865.6,-96906.59,'Bom','Sem Abrigo','No','Carris','776','386','Rua Sacadura Cabral','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
carreira('474',-106880.09,-96852.94,'Bom','Sem Abrigo','No','Carris','776','386','Rua Sacadura Cabral','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
carreira('463',-106886.32,-96345.37,'Bom','Sem Abrigo','No','Carris','776','386','Rua Sacadura Cabral','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
carreira('493',-106898.93,-96325.82,'Bom','Fechado dos Lados','No','Carris','776','369','Rua Direita do Dafundo','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
carreira('794',-106975.22,-95602.61,'Bom','Sem Abrigo','No','Carris','776','118','Alameda Hermano Patrone','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
carreira('144',-106979.51,-95226.45,'Bom','Fechado dos Lados','Yes','Carris','776','103','Rua Damiao de Gois','Alges, Linda-a-Velha e Cruz Quebrada-Dafundo').
```

Figura 4.2: Excerto da base de conhecimento com as carreiras

4.2 Criação dos grafos

Visto que as várias paragens de autocarros estão interligadas de acordo com a(s) carreira(s) que lhe(s) corresponde(m) e o objetivo deste trabalho é aplicar algoritmos de procura, achei por bem criar vários grafos de acordo com o que era pedido em cada tarefa.

4.2.1 Grafo geral

Assim sendo, comecei por criar o grafo geral que se trata de um grafo constituído por duas listas. Na primeira lista, estão todos os Gid's das paragens de autocarros (nós do grafo), enquanto na segunda lista estão representadas todas as adjacências entre as mesmas (arestas do grafo).

Para obter os nós deste grafo, recorri ao seguinte comando:

```
cut -d "," -f1 base_conhecimento_paragens.pl | cut -d "(" -f2 | sed 's/paragem(//g' |
sed 's/\$/,/g'
```

```
gids(['79',
'593',
'499',
'494',
'480',
'957',
'366',
'365',
'357',
'336',
'334',
'251',
'469',
'462',
```

Figura 4.3: Excerto da lista de nós

Quanto às arestas do grafo, recorri a um *script* que, para cada folha de cálculo do ficheiro Excel contendo a lista de adjacências, criou uma aresta entre os Gid's presentes em cada duas linhas consecutivas.

```
arestas([
  aresta('183','791')
, aresta('791','595')
, aresta('595','182')
, aresta('182','499')
, aresta('499','593')
, aresta('593','181')
, aresta('181','180')
, aresta('180','594')
, aresta('594','185')
, aresta('185','89')
, aresta('89','107')
, aresta('107','250')
, aresta('250','261')
, aresta('261','597')
, aresta('597','953')
, aresta('953','609')
, aresta('609','242')
, aresta('242','255')
, aresta('255','604')
, aresta('604','628')
, aresta('628','39')
```

Figura 4.4: Excerto da lista de arestas

Por fim, para representar o grafo, apenas foi necessário dar-lhe um nome (neste caso, "g") e invocar tanto a lista de nós como a lista de arestas, tal como podemos ver na figura abaixo.

```
g(grafo(Pontos,Arestas)) :- gids(Pontos),arestas(Arestas).
```

Figura 4.5: Representação do grafo geral

4.2.2 Grafo com os operadores

A segunda e terceira tarefas propostas no enunciado envolvem o conhecimento dos operadores associados a cada paragem de autocarro. Assim sendo, a única alteração que tive de realizar ao grafo geral foi trocar a lista de nós por uma lista onde os nós se encontram associados ao respetivo operador.

Para tal, recorri ao seguinte comando:

```
cut -d "," -f1,7 base_conhecimento_paragens.pl | cut -d "(" -f2 | sed 's/paragem(//g'
| sed 's/^/(/g' | sed 's/\$/),/g'
```

```
operadoras([('79','Vimeca'),
('593','Vimeca'),
('499','Vimeca'),
('494','Vimeca'),
('480','Vimeca'),
('957','Vimeca'),
('366','Vimeca'),
('365','Vimeca'),
('357','Vimeca'),
('336','Vimeca'),
('334','Vimeca'),
('251','Vimeca'),
('469','Vimeca'),
('462','Vimeca'),
('44','Vimeca'),
('78','Vimeca'),
('609','Vimeca'),
```

Figura 4.6: Excerto da lista de nós com os operadores associados

Por fim, para representar o grafo, apenas foi necessário dar-lhe um nome (neste caso, "g_op") e invocar tanto a lista de nós com os operadores associados como a lista de arestas, tal como podemos ver na figura abaixo.

```
g_op(grafo(Pontos,Arestas)) :- operadoras(Pontos),arestas(Arestas).
```

Figura 4.7: Representação do grafo com os operadores

4.2.3 Grafo com as carreiras

A quarta tarefa proposta no enunciado envolve conhecer as carreiras associadas a cada paragem. Assim sendo, a única alteração que tive de realizar ao grafo geral foi trocar a lista de nós por uma lista onde os nós se encontram associados à respetiva lista de carreiras a que pertencem.

Para tal, recorri ao seguinte comando:

```
cat base_conhecimento_paragens.pl | cut -d "(" -f2 | cut -d ")" -f1 | cut -d "]" -f 1
| sed 's/\$/]/g' | sed "s/',/'|/g" | cut -d"|" -f1,6 | sed 's/|/,/g' | sed 's/^/(/g' |
sed 's/\$/),/g'
```

```
carreiras([('79',[01]),
('593',[01]),
('499',[01]),
('494',[01]),
('480',[01]),
('957',[01]),
('366',[01]),
('365',[01]),
('357',[01]),
('336',[01]),
('334',[01]),
('251',[01]),
('469',[01]),
('462',[01]),
('44',[01,13,15]),
('78',[01,02,06,14]),
('609',[01,02,07,10,12,13,15]),
('599',[01,02,07,10,12,13,15]),
('595',[01,02,07,10,12,13,15]),
('185',[01,02,07,10,12,13,15]),
('250',[01,02,07,10,12,13,15]),
('107',[01,02,07,10,12,13,15]),
('953',[01,02,07,10,12,13,15]),
('594',[01,02,07,10,12,13,15]),
('597',[01,02,07,10,12,13,15]),
('261',[01,02,10]),
('341',[01,02,11]),
```

Figura 4.8: Excerto da lista de nós com a lista de carreiras associada

Por fim, para representar o grafo, apenas foi necessário dar-lhe um nome (neste caso, "g_carreiras") e invocar tanto a lista de nós com a lista de carreiras associada como a lista de arestas, tal como podemos ver na figura abaixo.

```
g_carreiras(grafo(Pontos,Arestas)) :- carreiras(Pontos),arestas(Arestas).
```

Figura 4.9: Representação do grafo com as carreiras

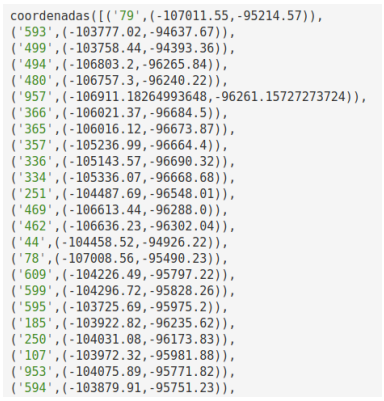
4.2.4 Grafo com as coordenadas

A sexta tarefa proposta no enunciado envolve as coordenadas de cada paragem. Assim sendo, a única alteração que tive de realizar ao grafo geral foi trocar a lista de nós por uma lista onde os nós se

encontram associados ao respetivo par de coordenadas (Latitude, Longitude).

Para tal, recorri ao seguinte comando:

```
cat base_conhecimento_paragens.pl | cut -d "(" -f2 | cut -d ")" -f1 | cut -d "]" -f 1
| sed 's/\$/]/g' | sed "s/',/'|/g" | sed -r "s/([0-9]),'/\1|'/g" | cut -d"|" -f1,2 | sed
"s/|/,(/" | sed "s/\$/)),/g" | sed "s/^/(/g"
```



```
coordenadas([('79',(-107011.55,-95214.57)),
('593',(-103777.02,-94637.67)),
('499',(-103758.44,-94393.36)),
('494',(-106803.2,-96265.84)),
('480',(-106757.3,-96240.22)),
('957',(-106911.18264993648,-96261.15727273724)),
('366',(-106021.37,-96684.5)),
('365',(-106016.12,-96673.87)),
('357',(-105236.99,-96664.4)),
('336',(-105143.57,-96690.32)),
('334',(-105336.07,-96668.68)),
('251',(-104487.69,-96548.01)),
('469',(-106613.44,-96288.0)),
('462',(-106636.23,-96302.04)),
('44',(-104458.52,-94926.22)),
('78',(-107008.56,-95490.23)),
('609',(-104226.49,-95797.22)),
('599',(-104296.72,-95828.26)),
('595',(-103725.69,-95975.2)),
('185',(-103922.82,-96235.62)),
('250',(-104031.08,-96173.83)),
('107',(-103972.32,-95981.88)),
('953',(-104075.89,-95771.82)),
('594',(-103879.91,-95751.23)),
```

Figura 4.10: Excerto da lista de nós com o par de coordenadas associado

Por fim, para representar o grafo, apenas foi necessário dar-lhe um nome (neste caso, "g_distancias") e invocar tanto a lista de nós com o par de coordenadas associado como a lista de arestas, tal como podemos ver na figura abaixo.

```
g_distancias(grafo(Pontos,Arestas)) :- coordenadas(Pontos),arestas(Arestas).
```

Figura 4.11: Representação do grafo com as coordenadas

4.2.5 Grafo com a indicação se existe publicidade

A sétima tarefa proposta no enunciado envolve saber se uma paragem tem ou não publicidade. Assim sendo, a única alteração que tive de realizar ao grafo geral foi trocar a lista de nós por uma lista onde os nós se encontram associados à respetiva indicação se existe publicidade ou não ("Yes"ou "No", respetivamente).

Para tal, recorri ao seguinte comando:

```
cut -d "," -f1,6 base_conhecimento_paragens.pl | cut -d "(" -f2 | sed 's/paragem(//g'
| sed 's/^/(/g' | sed 's/\$/),/g'
```

```
publicidade([('79','Yes'),
('593','No'),
('499','Yes'),
('494','No'),
('480','No'),
('957','No'),
('366','Yes'),
('365','Yes'),
('357','Yes'),
('336','Yes'),
('334','Yes'),
('251','Yes'),
('469','Yes'),
('462','No'),
('44','Yes'),
('78','Yes'),
('609','Yes'),
```

Figura 4.12: Excerto da lista de nós com a indicação se existe publicidade associada

Por fim, para representar o grafo, apenas foi necessário dar-lhe um nome (neste caso, "g_publicidade") e invocar tanto a lista de nós com a indicação se existe publicidade associada como a lista de arestas, tal como podemos ver na figura abaixo.

```
g_publicidade(grafo(Pontos,Arestas)) :- publicidade(Pontos),arestas(Arestas).
```

Figura 4.13: Representação do grafo com a indicação se existe publicidade

4.2.6 Grafo com os tipos de abrigo

A oitava tarefa proposta no enunciado envolve saber se uma paragem está ou não abrigada. Assim sendo, a única alteração que tive de realizar ao grafo geral foi trocar a lista de nós por uma lista onde os nós se encontram associados à respetiva condição do abrigo ("Sem Abrigo", "Fechado dos Lados"ou "Aberto dos Lados").

Para tal, recorri ao seguinte comando:

```
cut -d "," -f1,5 base_conhecimento_paragens.pl | cut -d "(" -f2 | sed 's/paragem(//g'
| sed 's/^/(/g' | sed 's/\$/),/g'
```

```
abrigos([('79','Fechado dos Lados'),
('593','Sem Abrigo'),
('499','Fechado dos Lados'),
('494','Sem Abrigo'),
('480','Sem Abrigo'),
('957','Sem Abrigo'),
('366','Fechado dos Lados'),
('365','Fechado dos Lados'),
('357','Fechado dos Lados'),
('336','Fechado dos Lados'),
('334','Fechado dos Lados'),
('251','Fechado dos Lados'),
('469','Fechado dos Lados'),
('462','Sem Abrigo'),
('44','Fechado dos Lados'),
('78','Fechado dos Lados'),
('609','Fechado dos Lados'),
('599','Fechado dos Lados'),
('595','Fechado dos Lados'),
('185','Fechado dos Lados'),
```

Figura 4.14: Excerto da lista de nós com o tipo de abrigo associado

Por fim, para representar o grafo, apenas foi necessário dar-lhe um nome (neste caso, "g_abrigo") e invocar tanto a lista de nós com os tipos de abrigo associados como a lista de arestas, tal como podemos ver na figura abaixo.

```
g_abrigo(grafo(Pontos,Arestas)) :- abrigos(Pontos),arestas(Arestas).
```

Figura 4.15: Representação do grafo com os tipos de abrigo

4.3 Proposta de resolução das tarefas

4.3.1 Tarefa 1

Para calcular um trajeto entre dois pontos, comecei por criar o predicado **adjacente** que recebe dois nós e um grafo (neste caso, o grafo "g") e vê se esses dois nós são adjacentes nesse grafo, ou seja, se existe alguma aresta entre eles.

De seguida, criei o predicado **caminho** que recebe um grafo, um ponto inicial e um ponto final e devolve o trajeto entre esses dois pontos nesse grafo. Para tal, vai sucessivamente procurando os pontos adjacentes ao ponto inicial até chegar ao ponto final.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Calcular um trajeto entre dois pontos

adjacente(X,Y,grafo(_,Es)) :- member(aresta(X,Y),Es).
adjacente(X,Y,grafo(_,Es)) :- member(aresta(Y,X),Es).

caminho(G,A,B,P) :- caminho1(G,A,[B],P).

caminho1(_,A,[A|P1],[A|P1]).
caminho1(G,A,[Y|P1],P) :-
   adjacente(X,Y,G), \+ memberchk(X,[Y|P1]), caminho1(G,A,[X,Y|P1],P).
```

Figura 4.16: Predicados criados para resolver a tarefa 1

4.3.2 Tarefa 2

Para selecionar apenas algumas das operadoras de transporte para um determinado percurso, comecei por criar o predicado **differ** que recebe a primeira lista do grafo "g_op"e a lista de operadores que se pretende selecionar e devolve a lista de Gid's cujos operadores não estão na lista que foi passada como parâmetro.

De seguida, criei o predicado **remove_aresta** que recebe a segunda lista do grafo "g_op"e a lista devolvida pelo predicado **differ** e devolve apenas a lista com as arestas formadas pelos Gid's que não se encontram na última lista passada como parâmetro.

Mais tarde, desenvolvi o predicado **inclui_op** que recebe o grafo "g_op", a lista de operadores que se pretende selecionar e devolve o grafo apenas com as arestas formadas pelos Gid's cujos operadores se encontram nessa lista.

Por fim, formulei o predicado **trajeto_com_op** que recebe o grafo "g_op", um ponto inicial, um ponto final, a lista com os operadores desejados e devolve, então, o percurso entre esses dois pontos, selecionando apenas os operadores passados como parâmetro.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -  -
% Selecionar apenas algumas das operadoras de transporte para um determinado percurso

differ([], _, []).
differ([(A,X)|Tail], L2, Result):- member(X, L2), !, differ(Tail, L2, Result).
differ([(A,X)|Tail], L2, Result):- differ(Tail, L2, Result1), Result = [A|Result1].

remove_aresta([], _, []).
remove_aresta([aresta(A,B)|Tail], L2, Result):- member(A, L2), !, remove_aresta(Tail, L2, Result);
                                                member(B, L2), !, remove_aresta(Tail, L2, Result).
remove_aresta([aresta(A,B)|Tail], L2, [aresta(A,B)|Result]):- remove_aresta(Tail, L2, Result).

inclui_op(G,[],[]).
inclui_op(grafo(A,B),L,R) :- differ(A,L,R1), remove_aresta(B,R1,R2), R = grafo(A,R2).

trajeto_com_op(G,A,B,O,P) :- inclui_op(G,O,R), caminho(R,A,B,P).
```

Figura 4.17: Predicados criados para resolver a tarefa 2

4.3.3 Tarefa 3

Para excluir um ou mais operadores de transporte para o percurso, comecei por criar o predicado **equals** que recebe a primeira lista do grafo "g_op"e a lista de operadores que se pretende excluir e devolve a lista de Gid's cujos operadores estão na lista que foi passada como parâmetro.

De seguida, voltei a usar o predicado **remove_aresta** que recebe a segunda lista do grafo "g_op"e a lista devolvida pelo predicado **equals** e devolve apenas a lista com as arestas formadas pelos Gid's que não se encontram na última lista passada como parâmetro.

Mais tarde, desenvolvi o predicado **exclui_op** que recebe o grafo "g_op", a lista de operadores que se pretende excluir e devolve o grafo apenas com as arestas formadas pelos Gid's cujos operadores não se encontram nessa lista.

Por fim, formulei o predicado **trajeto_sem_op** que recebe o grafo "g_op", um ponto inicial, um ponto final, a lista com os operadores indesejados e devolve, então, o percurso entre esses dois pontos, excluindo os operadores passados como parâmetro.

```
%------------------------------- - - - - - - - - - -  -  -  -  -   -
% Excluir um ou mais operadores de transporte para o percurso

equals([], _, []).
equals([(A,X)|Tail], L2, Result):- member(X, L2), !, equals(Tail, L2, Result1), Result = [A|Result1].
equals([(A,X)|Tail], L2, Result):- equals(Tail, L2, Result).

exclui_op(G,[],G).
exclui_op(grafo(A,B),L,R) :- equals(A,L,R1), remove_aresta(B,R1,R2), R = grafo(A,R2).

trajeto_sem_op(G,A,B,O,P) :- exclui_op(G,O,R), caminho(R,A,B,P).
```

Figura 4.18: Predicados criados para resolver a tarefa 3

4.3.4 Tarefa 4

Para identificar quais as paragens com o maior número de carreiras num determinado percurso, comecei por criar o predicado **devolve_lista_carreiras** que recebe o grafo "g_carreiras"e um ponto e devolve a lista de carreiras desse mesmo ponto.

De seguida, criei o predicado **comprimento_lista_carreiras** que recebe o grafo "g_carreiras"e um ponto e devolve um par formado por esse mesmo ponto e pelo comprimento da sua lista de carreiras.

Mais tarde, desenvolvi o predicado **obtem_comprimentos_listas_carreiras** que recebe o grafo "g_carreiras"e a lista com o nós que formam o percurso e devolve uma lista formada pelos pares (nó, tamanho da lista de carreiras).

Posteriormente, procedi à geração dos predicados **cabeca**, **segundo_elemento** e **segundo_da_cabeca**. O primeiro recebe uma lista e devolve o primeiro elemento da mesma, o segundo recebe um par e devolve o segundo elemento do mesmo e o terceiro recebe uma lista e devolve o segundo elemento do par que está à cabeça.

Depois, gerei o predicado **lista_maiores** que recebe a lista devolvida pelo predicado mencionado no terceiro parágrafo e devolve a lista com os pares que possuem o segundo elemento maior.

Ainda elaborei o predicado **lista_primeiro_elem** que recebe a lista devolvida pelo predicado anterior e devolve apenas os primeiros componentes de cada par.

Por fim, produzi o predicado **max_carreiras** que recebe o grafo "g_carreiras", um ponto inicial e um ponto final e devolve, então, as paragens com o maior número de carreiras no percurso entre o ponto inicial e o ponto final.

```
%--------------------------------- - - - - - - - - - -  -  -  -  -  -
% Identificar quais as paragens com o maior número de carreiras num determinado percurso

max_carreiras(G,A,B,L) :- caminho(G,A,B,P), obtem_comprimentos_listas_carreiras(G,P,R1), lista_maiores(R1,R2), lista_primeiro_elem(R2,L).

lista_primeiro_elem([],[]).
lista_primeiro_elem([(X,Y)|T],R) :- lista_primeiro_elem(T,R1), R = [X|R1].

lista_maiores([],[]).
lista_maiores([(X,Y) | T], R) :-
   lista_maiores(T, M), segundo_da_cabeca(M, P), Y > P, R = [(X, Y)].
lista_maiores([(X,Y) | T], R) :-
   lista_maiores(T, M), segundo_da_cabeca(M, P), Y == P, append(M, [(X, Y)], R).
lista_maiores([(X,Y) | T], R) :-
   lista_maiores(T, M), segundo_da_cabeca(M, P), Y < P, R = M.

segundo_da_cabeca(L, R) :- cabeca(L, P), segundo_elemento(P, R).

segundo_elemento((X,Y),Y).

cabeca([], (x, 0)).
cabeca([H|T], H).

obtem_comprimentos_listas_carreiras(G, [], []).
obtem_comprimentos_listas_carreiras(G, [H|T], L) :- comprimento_lista_carreiras(G,H,A), obtem_comprimentos_listas_carreiras(G,T,B), append([A],B,L).

comprimento_lista_carreiras(G,P,R) :- devolve_lista_carreiras(G,P,(X,R1)), length(R1,Res), R = (P,Res).

devolve_lista_carreiras(grafo([(X,Y)|T],B),P,R) :- P == X, R = (P,Y);
                                                    devolve_lista_carreiras(grafo(T,B),P,R).
```

Figura 4.19: Predicados criados para resolver a tarefa 4

4.3.5 Tarefa 5

Para escolher o menor percurso (usando critério menor número de paragens), comecei por criar o predicado **adjacentes** que recebe um ponto e um grafo (neste caso, o grafo "g") e devolve uma lista com todos os pontos adjacentes ao ponto passado como parâmetro no grafo fornecido.

De seguida, criei o predicado **caminhos** que recebe um ponto inicial, um ponto final, um grafo e a lista dos pontos adjacentes ao ponto inicial e devolve uma lista formada pelos pares de caminhos desde a cabeça da lista fornecida até ao ponto final e o respetivo comprimento.

Depois, desenvolvi o predicado **lista_menores** que recebe a lista resultante do predicado anterior e devolve a lista com os pares que possuem o segundo elemento menor.

Por último, produzi o predicado **caminho_com_menor_nr_paragens** que recebe um grafo, um ponto inicial e um ponto final e devolve, então, o menor percurso desde o ponto inicial até ao ponto final, ou seja, aquele com o menor número de paragens.

```
%--------------------------------- - - - - - - - - - -  -  -  -  -  -
% Escolher o menor percurso (usando critério menor número de paragens)

caminho_com_menor_nr_paragens(G,I,F,R) :- adjacentes(I,G,L1), caminhos(I,F,G,L1,L2), lista_menores(L2,[(A,B)|T]), append([I],A,R);
                                          caminho(G,I,F,R).

lista_menores([],[]).
lista_menores([X],[X]).
lista_menores([(X,Y) | T], R) :-
   lista_menores(T, M), segundo_da_cabeca(M, P), Y < P, R = [(X, Y)].
lista_menores([(X,Y) | T], R) :-
   lista_menores(T, M), segundo_da_cabeca(M, P), Y == P, append(M, [(X, Y)], R).
lista_menores([(X,Y) | T], R) :-
   lista_menores(T, M), segundo_da_cabeca(M, P), Y > P, R = M.

caminhos(I,F,G,[],[]).
caminhos(I,F,G,[H|T],R) :- caminho(G,H,F,L1), caminhos(I,F,G,T,L2), length(L1,R1), append([(L1,R1)],L2,R).

adjacentes(P,G,R) :- findall(X,adjacente(X,P,G),R).
```

Figura 4.20: Predicados criados para resolver a tarefa 5

4.3.6 Tarefa 6

Para escolher o percurso mais rápido (usando critério da distância) é desde já importante referir que, para tal, usei as coordenadas de localização (latitude e longitude) de cada paragem e determinei o

intervalo entre as mesmas, usando a distância euclidiana entre dois pontos, assumindo uma função tempo com base nessa distância (neste caso, 1 minuto por cada quilómetro).

Assim sendo, comecei por criar os predicados **distancia_entre_pontos** e **distancia_total**. O primeiro recebe dois pares de coordenadas e devolve a distância entre os mesmos. O segundo recebe uma lista com pares de coordenadas e devolve a soma da distância entre os mesmos.

De seguida, desenvolvi o predicado **coordenadas** que recebe o grafo "g_distancias"e um ponto e devolve as coordenadas do último no grafo passado como parâmetro.

Mais tarde, produzi o predicado **caminhos_com_distancia** que recebe um ponto inicial, um ponto final, o grafo "g_distancias"e a lista gerada pelo predicado **adjacentes** e devolve a lista com os pares formados pelos caminhos desde a cabeça da lista passada como parâmetro até ao ponto final e as respetivas distâncias totais, ou seja, incluíndo o ponto inicial.

Por último, gerei o predicado **caminho_com_menor_distancia** que recebe o grafo "g_distancias", um ponto inicial e um ponto final e devolve, então, o percurso mais rápido entre os dois pontos fornecidos, ou seja, aquele com a menor distância.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -  -
% Escolher o percurso mais rápido (usando critério da distância)

caminho_com_menor_distancia(G,I,F,R) :- adjacentes(I,G,L1), caminhos_com_distancia(I,F,G,L1,L2), lista_menores(L2,[(A,B)|T]), append([I],A,R);
                                        caminho(G,I,F,R).

caminhos_com_distancia(I,F,G,[],[]).
caminhos_com_distancia(I,F,G,[H|T],R) :- caminho(G,H,F,L1), caminhos_com_distancia(I,F,G,T,L2),
   maplist(coordenadas(G),L1,Y), distancia_total(Y,R1),
   coordenadas(G,I,CI), coordenadas(G,H,CH), distancia_entre_pontos(CI,CH,RC), Result is R1 + RC,
   append([(L1,Result)],L2,R).

coordenadas([],X,(_,_)).
coordenadas(grafo([(A,B)|T],C),X,R) :- X == A, R = B;
                                       coordenadas(grafo(T,C),X,R).

distancia_total([],0).
distancia_total([X],0).
distancia_total([X,Y|T],R) :- distancia_entre_pontos(X,Y,R1), distancia_total([Y|T],R2), R is R1+R2.

distancia_entre_pontos((X1,Y1),(X2,Y2),R) :- R1 = (X1-X2), R2 = (Y1-Y2), R3 is exp(R1,2), R4 is exp(R2,2), R is sqrt(R3+R4).
```

Figura 4.21: Predicados criados para resolver a tarefa 6

4.3.7 Tarefa 7

Para escolher o percurso que passe apenas por abrigos com publicidade, procedi a um raciocínio muito semelhante ao da tarefa 3.

Uma vez que apenas desejamos as paragens com publicidade, ou seja, aquelas cujo campo "Abrigo com Publicidade"tem "Yes", a única diferença está na lista passada ao predicado **exclui_op**, que deixa de ser a lista de operadores que se pretende excluir e passa a ser a lista com a *string* "No".

Assim sendo, apenas tive de criar o predicado **trajeto_com_publicidade** que recebe o grafo "g_publicidade", um ponto inicial e um ponto final e devolve, então, o percurso entre os dois pontos fornecidos que apenas passa por abrigos com publicidade.

```
%---------------------------------- - - - - - - - - - -  -  -  -  -  -
% Escolher o percurso que passe apenas por abrigos com publicidade

trajeto_com_publicidade(G,A,B,P) :- exclui_op(G,['No'],R), caminho(R,A,B,P).
```

Figura 4.22: Predicado criado para resolver a tarefa 7

4.3.8 Tarefa 8

Para escolher o percurso que passe apenas por paragens abrigadas, procedi a um raciocínio muito semelhante ao da tarefa 3.

Uma vez que apenas desejamos as paragens abrigadas, ou seja, aquelas cujo campo "Tipo de Abrigo"tem "Fechado dos Lados"ou "Aberto dos Lados", a única diferença está na lista passada ao predicado **exclui_op**, que deixa de ser a lista de operadores que se pretende excluir e passa a ser a lista com a *string* "Sem Abrigo".

Assim sendo, apenas tive de criar o predicado **trajeto_com_abrigo** que recebe o grafo "g_abrigo", um ponto inicial e um ponto final e devolve, então, o percurso entre os dois pontos fornecidos que apenas passa por paragens abrigadas.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -  -
% Escolher o percurso que passe apenas por paragens abrigadas

trajeto_com_abrigo(G,A,B,P) :- exclui_op(G,['Sem Abrigo'],R), caminho(R,A,B,P).
```

Figura 4.23: Predicado criado para resolver a tarefa 8

4.3.9 Tarefa 9

Para escolher um ou mais pontos intermédios por onde o percurso deverá passar, criei o predicado **caminho_com_pontos_intermedios** que recebe um grafo (neste caso, o grafo "g"), um ponto inicial, a lista de pontos intermédios pelos quais o percurso deverá passar e um ponto final e calcula, então, o trajeto entre os dois pontos fornecidos, passando pelos pontos intermédios passados como parâmetro. Para tal, calcula o trajeto entre o ponto inicial e o ponto que está à cabeça da lista fornecida, seguido do trajeto entre o último e o segundo ponto da lista fornecida e assim sucessivamente até chegar ao ponto final.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -  -
% Escolher um ou mais pontos intermédios por onde o percurso deverá passar.

caminho_com_pontos_intermedios(G, PI, [], PF, Caminho)
    :- caminho(G, PI, PF, Caminho).

caminho_com_pontos_intermedios(G, PI, [P | Ps], PF, Caminho) :-
    caminho(G, PI, P, Caminho1),
    caminho_com_pontos_intermedios(G, P, Ps, PF, Caminho_Restante),
    tail(Caminho_Restante, T),
    append(Caminho1, T, Caminho).
```

Figura 4.24: Predicado criado para resolver a tarefa 9

4.4 Testes

De forma a comprovar que todas as tarefas foram bem sucedidas, decidi criar testes com alguns exemplos.

Para a tarefa 1, criei o predicado **teste_ponto1** que recebe o grafo "g", um ponto inicial e um ponto final e testa se o resultado é igual ao resultado de aplicar o predicado **caminho** aos mesmos parâmetros. Imprime no terminal "Sucesso"ou "Insucesso", consoante os resultados sejam iguais ou

não, respetivamente.

De seguida, criei o predicado **run(1)** que aplica o predicado referido acima a um exemplo em concreto (neste caso, ao ponto inicial '183' e ao ponto final '181', tal como podemos ver na figura abaixo).

```
teste_ponto1(G, P1, P2, R) :- write('Teste ponto 1: '),
    caminho(G, P1, P2, P), P == R,
    write('Sucesso \n'); write('Insucesso \n').

run(1) :-
   g(G),
   teste_ponto1(G,'183','181',['183','791','595','182','499','593','181']).
```

Figura 4.25: Predicados criados para testar a tarefa 1

Para as restantes tarefas foram criados testes seguindo o mesmo raciocínio.

Capítulo 5

Conclusões e Sugestões

Após a realização deste primeiro exercício, concluo que um sistema de recomendação de transporte público para o sistema de transportes do concelho de Oeiras, capaz de demonstrar as funcionalidades subjacentes à programação em lógica, recorrendo à linguagem de programação Prolog foi desenvolvido com sucesso. Este inclui diversos tipos de conhecimento como paragens, carreiras, arestas e pontos.

Foram ainda implementadas todas as funcionalidades solicitadas com vários exemplos e demonstrações de cada um deles. Os procedimentos criados foram sendo acompanhados com pequenos exemplos de aplicação para que a sua eficiência fosse comprovada.

Em futuros desenvolvimentos deste trabalho poderão ser admitidos mais tipos de procedimentos (escolher o percurso que passe apenas por paragens em bom estado de conservação, por exemplo), ser permitidas pesquisas mais refinadas na base de conhecimento, acrescentando novos predicados e recorrer ainda à extensão da programação em lógica.

A boa preparação e o estudo prévio do caso de estudo e da problemática da programação em lógica possibilitaram que todo o processo decorresse dentro da normalidade, sem que tenham surgido problemas de grande dimensão.

Capítulo 6

Referências

6.1 Referências bibliográficas

[1] ANALIDE, Cesar, NOVAIS, Paulo, NEVES, José
"Sugestões para a Redacção de Relatórios Técnicos"
Relatório Técnico, Departamento de Informática, Universidade do Minho, Portugal, 2011

[2] BRATKO, Ivan
"PROLOG: Programming for Artificial Intelligence"
United States, Pearson Education (us), 2000

6.2 Referências Eletrónicas

[3] Oeiras Valley: Paragens de Autocarro
Disponível em: http://dadosabertos.cm-oeiras.pt/dataset/paragens-de-autocarro
Acesso em: 14 maio 2020

[4] PROLOG
Disponível em: https://pt.wikibooks.org/wiki/Prolog
Acesso em: 14 maio 2020

[5] SICStus Prolog
Disponível em: https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/
Acesso em: 14 maio 2020